How to Translate Altivec to SSE

Session 110

These are confidential sessions—please refrain from filming or taking pictures.

Application Technologies

How to Translate Altivec to SSE

Ali Sazegari, Ph.D.
Manager, Vector and Numerics Group, CoreOS

http://developer.apple.com/wwdc2005/

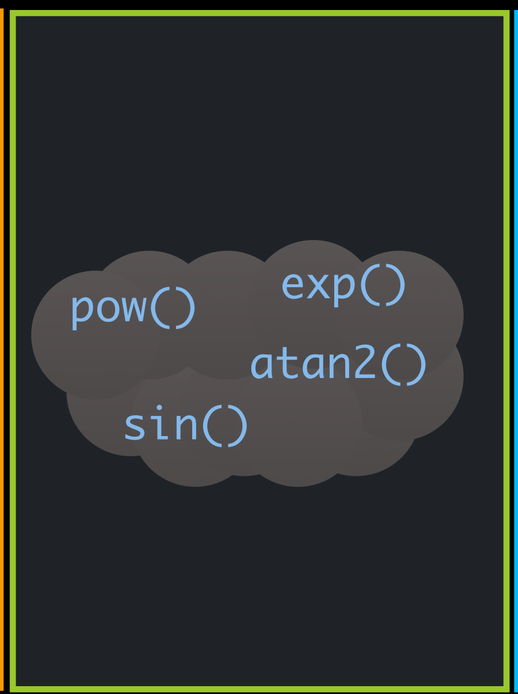
What You'll Learn

- What is the Accelerate framework?
- How to use Accelerate
- Translating Altivec to SSE
  - Hardware overview
  - Instruction overview
  - Translation techniques

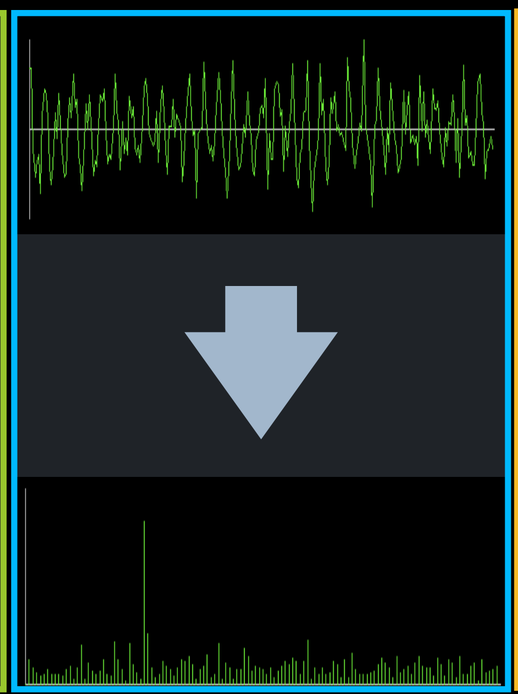
Strategies for Rapid Optimization on Mac OS X

Use Accelerate.framework:

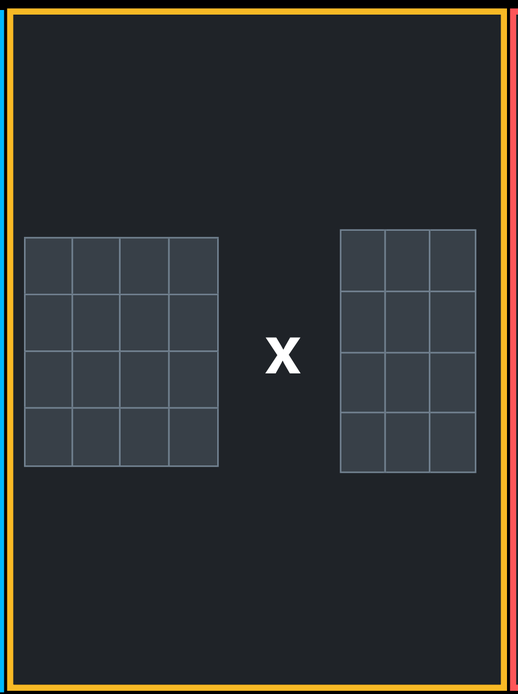
Image Processing
with vImage



Vector Math Libraries
vForce and vMathLib



Signal Processing
with vDSP



Linear Algebra
LAPACK and BLAS

Large Integer
Computation
vBasicOps and
vbigNum

Available for ppc, i386, and ppc64!

Accelerate.framework—BLAS and LAPACK

- Level 1, 2 and 3 BLAS
  - Vector—Vector operations
  - Vector—Matrix operations
  - Matrix—Matrix operations
- LAPACK
  - Solve systems of linear equations
  - Eigenvalue problems
  - Singular value problems
  - Dense and banded matrices

Accelerate.framework—vDSP

- Fourier Transforms
  - 1D and 2D
  - Real and complex
  - Single and double precision
- Fast Array Math
- Convolution and Correlation

Accelerate.framework—vMathLib and vForce

- vMathLib
  - SIMD (short vector) versions of common math library functions
    - Power, sine, cosine, exponential, logarithm, scalb, nextafter, etc.
    - For people writing vector code using AltiVec or SSE
- vForce
  - Array (long vector) versions of common math library functions
    - Power, sine, cosine, exponential, logarithm, scalb, nextafter, etc.
    - Useful for doing array operations in normal code—many times faster than libm
    - Callable from both C and Fortran
    - Automatically selects between scalar code and vector for best performance
    - You don’t have to do any vector programming

Accelerate.framework—vBasicOps and vBigNum

- vBasicOps
  - SIMD small integer multiplication, addition, shifts, division, modulus
  - Intended to round out vector C programming interfaces for common types
- vBigNum
  - SIMD large integer multiplication, addition, shifts, division, modulus
  - A fast, limited multi-precision library that uses AltiVec/SSE
    - Integers up to 1024 bits
    - Signed and unsigned

Accelerate.framework—vImage

- Vectorized image processing framework
  - Data types
    - 8-bit and floating point, mono-channel or packed ARGB
    - Does not need to be 16-byte aligned
  - Image operations
    - Convolution—blur, sharpen, emboss, edge detection, etc.
    - Morphology—reshape image elements, make them larger or smaller
    - Geometry—resize, rotate or distort images
    - Alpha—composite images in multiple layers with transparency
    - Conversions—convert among a variety of pixel packing formats
    - Transform—high-speed gamma correction and matrix multiplication
  - Automatically selects between vector and scalar code

FFT

Eric Postpischil
Senior Vector and Numerics Engineer, Vector and Numerics Group, CoreOS

http://developer.apple.com/wwdc2005/

Accelerate

Fast Fourier Transform (FFT), 1024 complex elements, on a G5 (2.7 GHz)

- Single precision

	Time (CPU cycles)	Time (microseconds)
Numerical Recipes	130,000	48.1
FFTW	32,300	11.9
Accelerate.framework	11,200	4.1

- Double precision

	Time (CPU cycles)	Time (microseconds)
Numerical Recipes	123,000	45.5
FFTW	40,400	14.9
Accelerate.framework	26,900	9.9

How to Use Accelerate.framework

- Include

```
#include <Accelerate/Accelerate.h>
```

- Link to
  - Xcode

    add `/System/Library/Frameworks/Accelerate.framework` to your project
  - Command line

    `-framework Accelerate`

How to Use Accelerate.framework's FFT

- Initialize the FFT once at the start of the application

```
//Do this only once!  DO NOT CALL FOR EACH FFT
  FFT_setup setup = vDSP_create_fftsetup( log2(length), kFFTRadix2 );
  if( NULL == setup )
        return error;
```

- Call the FFT

```
vDSP_fft_zip( setup, &data, I, log2(length), FFT_FORWARD );
```

- Clean up the FFT once at end of application

```
vDSP_destroy_fftsetup( setup );
```

Demo

FFT

Eric Postpischil
Senior Vector and Numerics Engineer, Vector and Numerics Group, CoreOS

http://developer.apple.com/wwdc2005/

Translating AltiVec to SSE*

Ian Ollmann, Ph.D.
Senior Scientist, Vector and Numerics Group, CoreOS

http://developer.apple.com/wwdc2005/

*** includes SSE, SSE2 and SSE3**

Translating AltiVec to SSE

- The good news

Translating AltiVec to SSE

- The good news
  - AltiVec and SSE are actually quite similar

Translating AltiVec to SSE

- The good news
  - AltiVec and SSE are actually quite similar
    - Same vector size (128 bits)
    - Pretty good instruction homology

Translating AltiVec to SSE

- The good news
  - AltiVec and SSE are actually quite similar
    - Same vector size (128 bits)
    - Pretty good instruction homology
  - In our experience, most of the work when vectorizing is parallelization, data layout and misalignment

Translating AltiVec to SSE

- The good news
  - AltiVec and SSE are actually quite similar
    - Same vector size (128 bits)
    - Pretty good instruction homology
  - In our experience, most of the work when vectorizing is parallelization, data layout and misalignment

…You've already done that work!

Translating AltiVec to SSE

- The good news
  - AltiVec and SSE are actually quite similar
    - Same vector size (128-bits)
    - Pretty good instruction homology
  - In our experience, most of the work when vectorizing is parallelization, data layout and misalignment

…You've already done that work!

- Expect translation of AltiVec to SSE to take a small fraction of the time it took to write the AltiVec in the first place, in most cases!

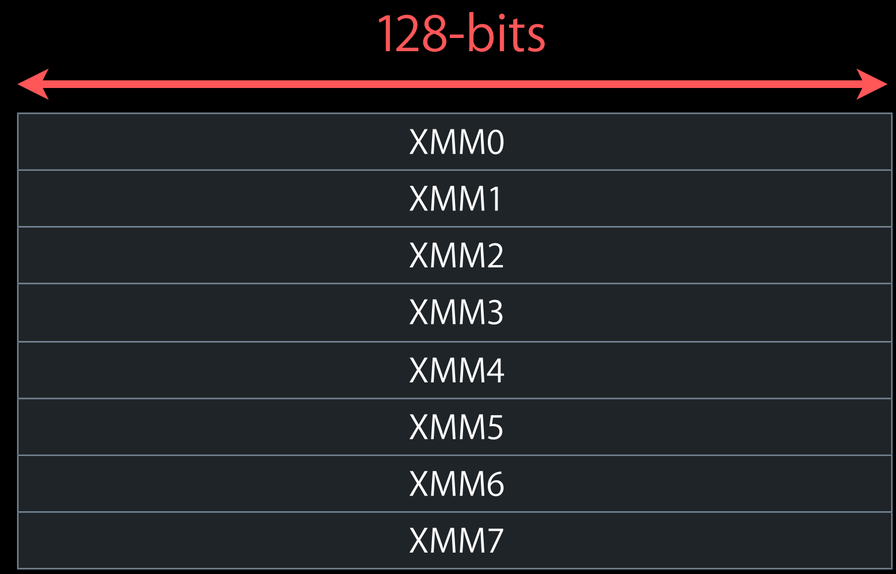
Intel's Streaming SIMD Extensions in Summary

- What you'll learn
  - Hardware overview
    - Registers, execution units and pipelines
    - Dispatch/issue
  - C programming model
    - Data Types
    - Intrinsics
  - AltiVec to SSE translation
    - How to do it
    - What to watch for

Hardware Overview (SSE)

Hardware Overview (SSE)

- Eight 128-bit XMM vector registers (16 for 64-bit)
- The register file is flat
  - Not stack-based like FPU
  - No special purpose registers like IU



- Instructions take no more than two register operands
- Instructions are typically destructive

Hardware Overview (SSE)

- Arithmetic/logical execution units (on the Developer Transition System)

FP_ADD	FP addition/subtaction, int<->float conversions, fp compares, min/max
FP_MUL	FP and integer multiplication
FP_DIV	Division and square root
FP_MISC	64-bit integer addition, certain permutes, some scalar loads and stores
FP_MOVE	Vector loads and stores, some scalar loads and stores
MMX_ALU	Integer addition, min/max. Integer and FP Boolean operations
MMX_SHFT	Nearly all permutes
MMX_MISC	Misc. conversions, floating point reciprocal estimate,

- Dispatch
  - Everything goes through port 1
  - …except FP_MOV, load and store (port 0, 2 and 3)

Instruction Overview (SSE)

- Supports scalar (FP only) and packed operations with

	8-bit	16-bit	32-bit	64-bit
Integer	ᗡ	ᗡ	ᗡ	ᗡ
Floating point			ᗡ	ᗡ

- Aligned and misaligned loads and stores
- Limited permute capability
- Predicated operation
- Saturated arithmetic
- Integer—multiply, add, subtract, min/max, SAD, Boolean
- FP—multiply, add, subtract, divide, square root, estimates, Boolean

C Programming Model (SSE)

C Programming Model (SSE)

- GCC and most other C compilers support the Intel C programming interface

C Programming Model (SSE)

- GCC and most other C compilers support the Intel C programming interface

…which is a lot like the AltiVec C programming interface

C Programming Model (SSE)

Data types

Intel on GCC

Any Packed Integer	Float[4]	Double[2]
__m128i	__m128	__m128d

Accelerate.framework types—work for both AltiVec and SSE!

	8-bit	16-bit	32-bit	64-bit
Signed	vSInt8	vSInt16	vSInt32	vSInt64
Unsigned	vUInt8	vUInt16	vUInt32	vUInt64
Floating point	—	—	vFloat	vDouble

Preferred for easier debugging!

C Programming Model (SSE)

Intrinsics

AltiVec

SSE

Notice that Accelerate.framework types are useful on both platforms

C Programming Model (SSE)

Intrinsics—no function overloading for SSE

AltiVec

```
vec_add( vUInt8, vUInt8)
```

SSE

```
_mm_add_epi8( vUInt8, vUInt8)
```

Notice that Accelerate.framework types are useful on both platforms

C Programming Model (SSE)

Intrinsics—no function overloading for SSE

AltiVec

```
vec_add( vUInt8, vUInt8)

vec_add( vUInt16, vUInt16)
```

SSE

```
_mm_add_epi8( vUInt8, vUInt8)

_mm_add_epi16( vUInt16, vUInt16)
```

Notice that Accelerate.framework types are useful on both platforms

C Programming Model (SSE)

Intrinsics—no function overloading for SSE

AltiVec

```
vec_add( vUInt8, vUInt8)

vec_add( vUInt16, vUInt16)

vec_add( vUInt32, vUInt32)
```

SSE

```
_mm_add_epi8( vUInt8, vUInt8)

_mm_add_epi16( vUInt16, vUInt16)

_mm_add_epi32( vUInt32, vUInt32)
```

Notice that Accelerate.framework types are useful on both platforms

C Programming Model (SSE)

Intrinsics—no function overloading for SSE

AltiVec

```
vec_add( vUInt8, vUInt8)

vec_add( vUInt16, vUInt16)

vec_add( vUInt32, vUInt32)

vec_add( vFloat, vFloat)
```

SSE

```
_mm_add_epi8( vUInt8, vUInt8)

_mm_add_epi16( vUInt16, vUInt16)

_mm_add_epi32( vUInt32, vUInt32)

_mm_add_ps( vFloat, vFloat)
```

Notice that Accelerate.framework types are useful on both platforms

C Programming Model (SSE)

Intrinsics—no function overloading for SSE

AltiVec

```
vec_add( vUInt8, vUInt8)

vec_add( vUInt16, vUInt16)

vec_add( vUInt32, vUInt32)

vec_add( vFloat, vFloat)
```

SSE

```
_mm_add_epi8( vUInt8, vUInt8)

_mm_add_epi16( vUInt16, vUInt16)

_mm_add_epi32( vUInt32, vUInt32)

_mm_add_ps( vFloat, vFloat)
```

also...

```
_mm_add_epi64( vUInt64, vUInt64)
_mm_add_pd( vDouble, vDouble)

_mm_add_ss( vFloat, vFloat)
_mm_add_sd( vDouble, vDouble)
```

Notice that Accelerate.framework types are useful on both platforms

C Programming Model (SSE)

Intrinsics

AltiVec

```
vec_add( vUInt8, vUInt8)

vec_add( vUInt16, vUInt16)

vec_add( vUInt32, vUInt32)

vec_add( vFloat, vFloat)
```

SSE

```
_mm_add_epi8( vUInt8, vUInt8)

_mm_add_epi16( vUInt16, vUInt16)

_mm_add_epi32( vUInt32, vUInt32)

_mm_add_ps( vFloat, vFloat)
```

also...

```
_mm_add_epi64( vUInt64, vUInt64)
_mm_add_pd( vDouble, vDouble)

_mm_add_ss( vFloat, vFloat)
_mm_add_sd( vDouble, vDouble)
```

Notice that Accelerate.framework types are useful on both platforms

C Programming Model (SSE)

Sample function—AltiVec

- Distance from origin for a set of four {X,Y} pairs: sqrt(x2 + y2)

```
#include <Accelerate/Accelerate.h>

vFloat Distance( vFloat x, vFloat y ){
    vFloat  x2 = vec_madd( x, x, (vector float) (-0.0f) );   // x2
    vFloat distance2 = vec_madd( y, y, x2 );         // x2 + y2

    return vsqrtf( distance2 );      //from Accelerate.framework
}
```

C Programming Model (SSE)

Sample function—SSE

- Distance from origin for a set of four {X,Y} pairs: sqrt(x2 + y2)

```
#include <Accelerate/Accelerate.h>
#include <xmmintrin.h>

vFloat Distance( vFloat x, vFloat y ){
    vFloat  x2 = _mm_mul_ps( x, x );   // x2
    vFloat distance2 = _mm_add_ps(_mm_mul_ps( y, y), x2 );    // x2 + y2

    return vsqrtf( distance2 );       //from Accelerate.framework
}
```

C Programming Model (SSE)

Sample function—both!

- Distance from origin for a set of four {X,Y} pairs: sqrt(x2 + y2)

```
#include <Accelerate/Accelerate.h>

vFloat Distance( vFloat x, vFloat y ){
    return vsqrtf( x*x + y*y );      //from Accelerate.framework
}
```

C Programming Model (SSE)

Detecting the vector unit

- Guaranteed to be there
  - MMX
  - SSE
  - SSE2
- To detect SSE3

```
#include <sys/sysctl.h>

int hasSSE3( void ){
    int hasSSE3 = 0;
    size_t length = sizeof( hasSSE3 );
    int error = sysctlbyname( “hw.optional.sse3”, &hasSSE3, &length, NULL, 0);
    if( 0 != error ) return 0;
    return hasSSE3;
}
```

Translating AltiVec to SSE

Translating AltiVec to SSE

Floating point

- Very good matchup in general—both are influenced by IEEE-754
- SSE is like AltiVec except that you also get
  - Double precision
  - Scalar operations (at double the throughput of packed)
  - Floating point divide and square root in hardware
  - Fully IEEE-754 compliant machine
- …but you do not get (AltiVec only)
  - Multiply-add fused—expect different answers!
  - Vector compare bounds
  - $Log_2()$ and 2^x estimates

Translating AltiVec to SSE

Floating point

- Use Accelerate.framework for transcendentals, as always
  - #include <Accelerate/Accelerate.h>
    vsinf(), vcosf(), vpowf(), vlogf(), vexpf(), vnextafterf(), etc.
- New! Full IEEE-754 support
  - Exceptions
  - Rounding modes
  - Flags
  - Check MXCSR for appropriate bits

```
int flagbits = _mm_getscr();
```

Translating AltiVec to SSE

Floating point

- Denormals
  - You can trap on denormals
  - Turn denormal handling on and off

	Intel	G5
By default, denorms are:	ON	OFF
Penalty for hitting one:	1500 cycles	1100 cycles

  - How to turn denormals off

```
#include <xmmintrin.h>
_mm_setcsr( _mm_getcsr() | 0x8040 );
```

Translating AltiVec to SSE

Integer

- Instructions derived from MMX
  - Typically old MMX instruction done twice in a XMM register
  - …64-bits at a time
- Add/subtract
  - Modulo—8, 16, 32 and 64-bit add/subtract
  - Saturated—8, 16 bit saturated add/subtract only

Translating AltiVec to SSE

Integer

- Min/max
  - Signed and unsigned
  - 8- and 16-bit only
- Multiplication—almost no overlap with AltiVec
  - PMULLW—16-bit multiply, low 16-bit result (a bit like vec_mladd)
  - PMULHW—16-bit signed multiply, high 16-bit result
  - PMULHUW—16-bit unsigned multiply, high 16-bits
  - PMADDWD—16-bit signed multiply and add (a bit like vec_msum)
  - PMULUDQ—32-bit unsigned multiply to give 64-bit results

Translating AltiVec to SSE

Compare

- Operate just like AltiVec, return -1 for true, 0 for false
- …but do not set CR

```
vec_cmpeq( vFloat, vFloat );        _mm_cmpeq_ps( vFloat, vFloat );
```

- Compares
  - Full complement of floating point compares
    - Scalar and packed
    - <, <=, ==, !=, >=, >, unordered, ordered
  - Integer compares
    - Signed only
    - No 64-bit integer compares

Translating AltiVec to SSE

Predication

- Conditional execution
  - Use MOVMSKPS, MOVMSKPD, PMOVMSKB to copy top bit of compare mask
  - Compare the bits against a constant

```
int vec_any_eq( vFloat a, vFloat b)
{
     vFloat mask = _mm_cmpeq_ps( a, b );
     int  maskbits = _mm_movemask_ps( mask );
     return maskbits != 0;
}
```

Translating AltiVec to SSE

Predication

- Select
  - There is no select instruction
  - Use AND, ANDNOT, OR: result = (a & ~mask) | (b & mask)

```
vFloat vec_sel( vFloat, vFloat, vUInt32 );


vFloat _mm_sel_ps( vFloat a, vFloat b, vUInt32 mask)
{
     a = _mm_andnot_ps( mask, a );    //first argument is the complement!
     b = _mm_and_ps( mask, b );
     return _mm_or_ps( a, b );
}
```

Translating AltiVec to SSE

Type conversions

- Float ⇔ int
  - Signed ints only, no unsigned support
  - Float ⇒ int: values >= 231 saturate to 0x80000000, not 0x7FFFFFFF

```
vSInt32 vec_cts( vFloat v )
{
    vFloat two31 = (vFloat) { 0x1.0p31, 0x1.0p31, 0x1.0p31, 0x1.0p31};
    vUInt32 overflow = _mm_cmpge_ps( v, two31 );
    vSInt32  result = _mm_cvtps_epi32( v );  //uses current rounding mode

    return _mm_xor_ps( result, overflow );   //flip 0x80000000 to 0x7FFFFFFF
}
```

Translating AltiVec to SSE

Type conversions

- Int ⇔ int
  - Large int ⇒ small int
    - Saturated pack
      - Signed: int ⇒ short and short ⇒char
      - Unsigned: short ⇒ unsigned char (no int ⇒ unsigned short)
    - No unsaturated pack
      - Truncate first, then use saturated pack

```
//unsaturated 32-bit ⇒ 16-bit pack  (shift left, shift right algebraic, packs)

_mm_packs_epi32( _mm_srai_epi32( _mm_slli_epi32( v, 16), 16));
```

      - Permute for 64-bit ⇒ 32-bit

Translating AltiVec to SSE

Type conversions

- Int ⇔ int
  - Small int ⇒ large int
    - Unsigned
      - Use unpack instructions, unpack with zero as with AltiVec
      - Available for 64-bit sizes too
    - No signed unpack support
      - Merge(a,a) then shift right algebraic

```
//signed 16-bit ⇒ 32-bit unpack hi (merge, shift right algebraic)

_mm_srai_epi32( _mm_unpackhi_epi16( v, v), 16);
```

      - Merge with cmplt(v, zero) for 64-bit

Translating AltiVec to SSE

Permute—unpack

- Bytes, words (16-bit), doublewords (32-bit), quadwords (64-bit)
- 4x4 transpose example:

```
//note: hi and lo are reversed from AltiVec
   vUInt32 t0 = _mm_unpacklo_epi32( a0, a2 );
   vUInt32 t1 = _mm_unpackhi_epi32( a0, a2 );
   vUInt32 t2 = _mm_unpacklo_epi32( a1, a3 );
   vUInt32 t3 = _mm_unpackhi_epi32( a1, a3 );
   a0 = _mm_unpacklo_epi32( t0, t2 );
   a1 = _mm_unpackhi_epi32( t0, t2 );
   a2 = _mm_unpacklo_epi32( t1, t3 );
   a3 = _mm_unpackhi_epi32( t1, t3 );
```

Translating AltiVec to SSE

Permute—shifts

- Available in left/right logical and right algebraic varieties

	8-bit	16-bit	32-bit	64-bit	128-bit
Left logical	None	P	P	P	P
Right logical	None	P	P	P	P
Right algebraic	None	P	P	None	None

- Two flavors
  - Shift by immediate—must be known at compile time
  - Shift by value—elements may not be shifted by different values
- No 8-bit shifts
- 128-bit shift by multiples of 8 bits only
- 128-bit shift by immediate only

Translating AltiVec to SSE

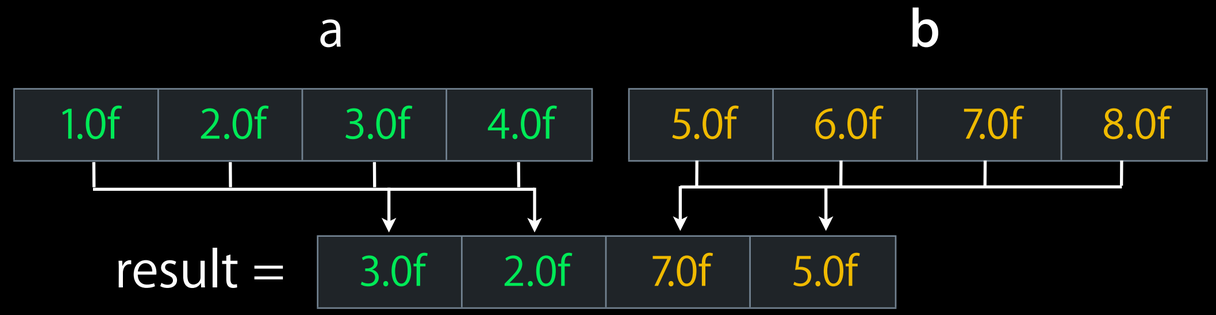
Permute—shuffle

- Like vec_perm
- Uses a constant defined at compile time for permute map
- Elements are the basic quantum, not bytes
- Shuffle maps (i)
  - Are always immediates
  - Four packed two-bit fields

```
    ( index0 << 6) | ( index1 << 4) | ( index2 << 2) |  index3
```

Translating AltiVec to SSE

Permute—shuffle types



- Float

  _mm_shuffle_ps(a, b, i);

- Int

  _mm_shuffle_epi32(a, i) any ⇒ any

- Short

  _mm_shuffle_epi16(a, i) any* ⇒ any*

- 8-bit

  None!

*values may not cross 64-bit segments

Translating AltiVec to SSE

Permute

- Many things cannot be done with permute
  - Sometimes difficult to move data between low and high 64-bits
  - Minimum permute level is word (16-bit), not byte
    - Byteswap—use 16-bit shifts and OR (v<<8) | (v >> 8)
    - Bytewise unpack is available
  - No data dependent data movement—immediate permute constants only
    - Lookup tables
    - Selecting pivot elements in register
    - Align/misalign data—no 128-bit shift by variable

Translating AltiVec to SSE

Loads and stores

- Scalar loads and stores
  - Always from element 0, using MOVSS, MOVSD or MOVD
  - Do not translate well to AltiVec due to alignment
- Vector loads and stores on SSE
  - Aligned loads and stores
    - Fast
    - **ACCESS EXCEPTION** if pointer is misaligned
  - Misaligned loads and stores
    - 1.5–2.0x slower, even if the pointer is aligned
    - Stores are 13x slower if < 8 byte aligned

Translating AltiVec to SSE

Loads and stores

- Misaligned access is handled completely differently; doesn't translate well
- Always align stores for performance
- Use scalar (or scalar on vector) code at the misaligned edges for stores
- Be careful of common crashing bugs
  - It is not safe to touch a misaligned vector unless *all* bytes are known to exist
  - Don't pass misaligned pointers to aligned loads and stores

Translating AltiVec to SSE

Loads and stores

- One more thing…

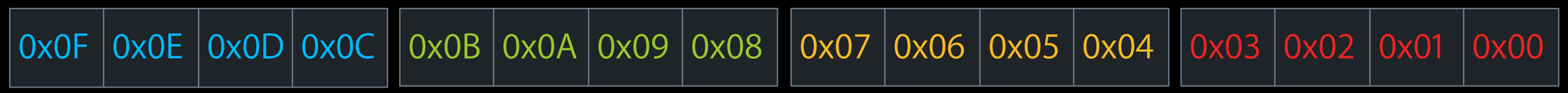
Translating AltiVec to SSE

Loads and stores

- One more thing...
  - Byteswapping on load/store reverses element order from memory

Order in memory

Order after loading to register



- Which means that:

{0.0f, 1.0f, 2.0f, 3.0f} appears in register as { 3.0f, 2.0f, 1.0f, 0.0f }!

Translating AltiVec to SSE

Unrolling

- For PowerPC, one frequently unrolled in parallel, exploiting 32 registers:

```
a0 += 4;        a1 += 4;        a2 += 4;        a3 += 4;
a0 *= b0;       a1 *= b1;       a2 *= b2;       a3 *= b3;
a0 = a0 >> 8;   a1 = a1 >> 8;   a2= a2 >> 8;    a3 = a3 >> 8;
...
```

- For SSE, there are not enough registers for that usually, so unroll serially*:

```
a += 4;
a *= b;
a = a >> 8;
(store a, load next a and b)
a += 4;
a *= b;
a = a >> 8;
....
```

- …which parallelizes in the reorder buffers using many rename registers

*but not on Developer Transition System. Serial unrolling wastes trace cache space

Translating AltiVec to SSE

Performance tips

- Reduce/eliminate your need for the permute unit
  - Change data layouts
  - Align data
- Do not unroll as much for parallelism (serial unrolling okay)
  - There aren't enough registers for that—you'll spill
  - The pipelines are short and half pumped—there isn't much to fill
- Don't waste time synthesizing constants
  - We don't have vec_splat_* here
  - They will just spill, so you will have to load them in on demand anyway

Sounds Hard?

Use the Accelerate.framework!

Intel's Streaming SIMD Extensions in Summary

What you've learned

- Hardware overview
  - Fewer registers, 64-bit ALU and shorter pipelines—unroll differently
  - Nearly all vector instructions dispatch through a single port
- C programming model
  - Data Types—use the Accelerate types for best debugging
  - Intrinsics—#include <xmmintrin.h> to use Intel intrinsics
- AltiVec to SSE translation
  - Some code can port over fairly cleanly, but not permute, misalignment, others
  - Element order in register is backward
  - Use Accelerate.framework to avoid porting worries

More Information

http://developer.apple.com/wwdc2005/

Documentation, Sample Code, and Other Resources

http://developer.apple.com/wwdc2005/

Reference Library

Velocity Engine Web Page

http://developer.apple.com/hardware/ve

Intel Reference Documents

http://www.intel.com/design/pentium4/manuals/index_new.htm

Universal Binary Programming Guidelines

http://developer.apple.com/documentation/MacOSX/Conceptual/universal_binary/universal_binary.pdf

http://tinyurl.com/augr5

Who to Contact

Xavier Legros
Mac OS X Technology Evangelist
xavier@apple.com

Matthew Formica
Developer Tools and Cocoa Evangelist
mformica@apple.com

Performance Tools Feedback List
perftools-feedback@group.apple.com

Q&A

http://developer.apple.com/wwdc2005/

WWDC2005

Worldwide Developers Conference